Lexical Analysis on a Moderately Sized Multiprocessor

James R. Low

Technical Report 261
October 1988

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Lexical Analysis on a Moderately Sized Multiprocessor[†]

TR 261

October 1988

James R. Low
Department Of Computer Science
The University of Rochester
Rochester, N. Y. 14627

Abstract

A method for using dozens of conventional VonNeuman processors to tokenize programs in parallel is described. Results of a prototype implementation on the BBN Butterfly Computer are discussed.

[†] This work was supported in part by NSF Coordinated Experimental Research grant no. DCR-8320136 and in part by U. S. Army Engineering Topographic Laboratories contract no. DACA76-85-C-0001. We thank the Xerox Corporation University Grants Program for providing equipment used in the preparation of this report.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TR 261	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: Lexical Analysis on a Moderately Sized Multiprocessor	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): James R. Low	8. CONTRACT OR GRANT NUMBER(s): DACA76-85-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science 734 Computer Studies Bldg. Univ. of Rochester, Rochester, NY 14627	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: DARPA/1400 Wilson Blvd. Arlington, VA 22209	12. REPORT DATE: October 1988 13. NUMBER OF PAGES: 22
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Office of Naval Research Information Systems Arlington, VA 22217	15. SECURITY CLASS. *(of this report)*: Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*
Distribution of this document is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*
Lexical Analysis
Multiprocessors
Butterfly
Parallel Programming
Compilers

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*
A method for using dozens of conventional VonNueman processors to tokenize programs in parallel is described. Results of a prototype implementation on the BBN Butterfly Computer are discussed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

1. INTRODUCTION

1.1. Moderately Sized Multiprocessors

In the past few years, several computing systems have been developed which use traditional Von-Neuman processors executing in parallel. These processors communicate at speeds which are only slightly slower than accesses to local memory. Communication is accomplished using message passing and/or shared memory techniques. These computers are described as MIMD (Multiple Instruction Multiple Data) machines. We use the term *Moderately Sized Multiprocessor* to describe MIMD systems which contain a moderate number (several dozen) processors. Examples of such systems include the Cal Tech Cosmic-Cube [17], the NYU UltraComputer [7], the IBM RP3 [15], and the BBN Butterfly Computer [3].

1.2. Compiling On/For a Multiprocessor

Two interesting problems in compiling come to our attention. The first (and admittedly the more important) problem involves building compilers to assist the programmer in building applications that will take advantage of the parallelism offered by the multiprocessor. This problem we designate as *Compiling For a Multiprocessor.* The other problem is how to build a compiler which executes simultaneously on several nodes of a multiprocessor even though the output of the compiler may be destined for a uniprocessor or a single node of a multiprocessor. This problem is designated as *Using a Multiprocessor for Compiling*. In this paper we address some of the issues involved with the second problem: using a multiprocessor to perform traditional compiling activities. We sketch the overall design of a Compiler using a Moderately Sized Multiprocessor. Our emphasis is on the scanning (lexical analysis) stage of compiling. We have built a prototype of a scanner for the programming language Modula 2 [21] which executes on a BBN Butterfly Computer [3]. Results using that prototype are discussed.

1.3. Overview of Traditional Compiler Architectures

Compilers are traditionally organized as a set of modules performing different portions of the compiling activity [1, 22]. One decomposition of a compiler might include; a *preprocessor* performing file inclusion and macro expansion; a *scanner* (lexical analysis) module to break an input stream of characters into tokens (lexemes); a *parser* (syntactic analysis module) that parses a token stream according to some grammar and whose output is a symbol table and a set of tuples or some other intermediate representation of the program; a *data flow analysis* module to derive relationships between expression computations and uses; an *optimizer* to remove redundant operations; a *code generator* producing an assembly or object module; and perhaps an *assembly* phase to generate the final object module. These modules may communicate in several

different ways. The modules may run sequentially in passes (with the results of each pass being an intermediate data structure or data file); as pipelines where data produced by one module is passed to the next as it is produced; or as coroutines executing in a pseudo-parallel manner.

Note that this traditional compiler architecture has been challenged in recent years by the use of programming environments containing editors which manipulate data structures representing the parse tree of the program instead of a character string representation of the program (e.g INTERLISP [19], and the Cornell Program Synthesizer [18]). In these architectures, the compiler does not perform the scanning or parsing phases as these have essentially been completed by the editor. Our work here does not consider such programming environments.

1.4. Gross Parallelism

One of the easiest ways of using a multiprocessor to get leverage on the compiling problem is to assume that we have a large number of different source files to compile. We then run a traditional compiler on each node of the multiprocessor. Thus, if we had one hundred processing nodes we would have one hundred compilers running in parallel. This is exemplified in the parallel versions of the Unix(tm) *make* facility [4]. This would (assuming sufficient I/O bandwidth) give us a near linear speed-up in compilation time for the set of programs to be compiled. However, in the course of application program development, it is rare to have that many files to compile at once. Subsystems that must be recompiled together consist of relatively few source files. For example, the Chrysalis Operating System [2] for the BBN Butterfly Computer currently is made up of only a few dozen files. In its extreme, the gross parallelism technique would encourage us to construct source files containing single procedures, to allow us to maximize the parallelism in the compilation. This changes the problem from parallel compiling to parallel linking. The number of source files that a programmer has to keep track of would also increase dramatically. The gross parallelism approach would not give us much leverage in the edit-compile-debug loop as we would typically have only one or two source files modified at a time. Thus, we would probably only use a very small number of the available processors when we did compile. However, if we were recompiling all of a very large system (say on operating system and all the dependent utilities) this might be the best approach to follow.

1.5. Pipelining

We may use a few processors by dedicating each to a particular phase of the compilation. The stream of output from each processor becomes the input to the next processor in the pipeline. One processor might be assigned to the macro-expansion; another the scanner; another the parser; and so forth. Some processors are idle at the beginning of a compilation while the pipeline is being filled, but during most of the compilation, useful work is done by all. Care must

be taken so that comparable amounts of computation would be done by all processors to avoid excessive idle times among individual processors waiting for input from a previous processor in the pipeline. With the straightforward implementation of this technique, it is hard to imagine how more than ten or so processors could be used. This technique could be combined with the Gross Parallelism technique mentioned above to increase the number of processors used.

1.6. Parallel Implementations of the Passes

Our current research involves investigation of each of the phases of the compiler and development of parallel algorithms to simultaneously use several processors for each phase. Our current model assumes that each phase of the compilation run to completion before the next is started . This unfortunately eliminates pipelining or coroutining of the stages. We expect it will be possible to relax this restriction in the future.

1.7. Evaluation of a Parallel Implementation

An evaluation of a parallel implementation may involve many quantities: memory use; execution time; and so forth. Our evaluation is in terms of execution time. For ease of comparison let us assume that we are comparing a multiprocessor and uniprocessors with sufficient memory and memory addressibility so that issues of addressibility do not cloud a comparison of execution times. We do not want to consider problems that "fit" in main memory on a multiprocessor but must be "swapped" in and out on a uniprocessor because of limited addressibility or real memory directly available to a single processor.

What should be the execution time of a multiprocessor implementation be relative to that of a uni-processor implementation? Let us define a few terms so that we may discuss execution times.

For a given problem let:

> Time(n, m) be the time needed to process the first *m* items of data using *n* processors. In other words the time needed by *n* processors to handle a problem of size *m*. Clearly this is data file dependent, but let us consider this time for a particular data set.

Our goal is to get a speedup of *n* times using *n* processors.

> Time(n,m) = Time(1,m) / n

In general this is not possible, for several reasons.

First of all there is a non-zero overhead in starting and finishing up that is independent of

problem size. In other words

$$\text{Time}(1, 0) > 0$$

There is also a difference in the execution time needed for processing portions of the input file. For example, in scanning, a processor which receives one long comment as its input will finish significantly sooner than a processor which receives normal statements. In scanning, we have seen factors of 3 differences in execution time to process different program segments that are approximately the same length. This problem we call the problem of *non-uniform partitioning*, as the amount of computation is non-uniform across the partitions.

Finally, there is the problem of contention. There may be some critical resource either in the hardware or in the data structures of the program that prevents all processors executing concurrently. An example of such a hardware resource is the memory switch of the Butterfly Computer. Most data structures that maintain consistency by allowing multiple readers or a single writer will be potential contention points.

2. RELATED WORK

Related work includes the pipelined compiler of Miller and Leblanc [11]. They were limited in the amount of parallelism by the length of their pipeline. Their pipeline consisted of a scanning stage, a parsing stage and a semantics stage.

Vandervoorde's compiler for C [20] basically ignored parallel scanning and concentrated on parallel instantiations of a recursive descent parser, with a new parser thread being forked when a new code (as opposed to declaration) section is encountered. He describes techniques to keep the number of threads comparable to the number of available processors to avoid the overhead of thread context switching. His work was on a small multiprocessor (5 processors) and he was able to get a speed up of over 3 compared with a uniprocessor implementation.

Mikunas and Schell [13] describe a design for a parallel parser in which ambiguities by one processor are disambiguated using communication with the processor to the left. Apparently no implementation was completed.

In summary, most earlier work seems to have concentrated on utilization of a relatively small number of processors (less than 10). Simple pipelining techniques or techniques involving potential communications with a small number of other processors have been previously demonstrated.

3. PARALLEL IMPLEMENTATIONS OF THE PASSES

Our model of a parallel compiler is to parallelize each stage of compilation, running each stage to completion before starting the next stage. In this section we present the overall design of a parallel compiler.

3.1. Scanning

We partition the source file into arbitrary source-pieces (say every few hundred bytes) and hand off each source-piece to a different processor and then concatenate the resulting token streams. The main problem with this technique is that apriori we have no way of insuring that the partitioning points are between tokens. If we picked totally arbitrary partitioning points we could be in the middle of a reserved word, identifier, multicharacter operator, comment or character literal. Solutions to this problem will be discussed below.

The output of the scanner consists of several data structures: the table of the tokens found; a string table containing all character literals and names of identifiers; some description about the comments terminated in this source-piece (but not started) in this piece; a description of the comments started but not terminated; and some indication of the bracketing constructs (e. g. **PROCEDURE** tokens and their corresponding **END** tokens) which would be useful to the next stage to partition the token stream into suitable groups for parceling out to parallel instantiations of a parser.

3.2. Parsing

Using the bracketing information (for example **PROCEDURE** and corresponding **END**) to identify complete syntactic units, we partition the token stream into groups. Each of the groups of tokens derived above is sent to a different instantiation of the Parsing module. There may be different types of Parsers depending on the known syntactic unit. E. g. one type of parser for Module bodies, one type for procedures etc. The output of each parser instantiation is the intermediate code form (e. g. parse tree or tuples) and a symbol table segment. We have no guarantee that the parsing of a Module body or Procedure will be completed before Procedures nested inside the Module body. Thus, we cannot resolve symbol references during this stage. The intermediate code will thus have references to the string table for all identifier usages.

3.3. Symbol Table Construction

We merge the symbol table segments produced above and build a data structure reflecting the scoping rules. In Modula-2 we would also have to merge in the symbol table segments produced by previous compilations of **DEFINITIONS** modules at this time.

3.4. Name Resolution

Because of the possible nesting and hiding of declarations it is not possible in general for parallel parsers to determine the binding between identifier usages and declarations until after the Symbol Table has been constructed (above).

Therefore we need a phase which processes the intermediate code (may coincide with the code generation phase) to map the identifiers into the appropriate symbol table entries.

3.5. Other Phases

The optimization, code generation and assembly phase (if any) could be done in parallel as well (say on a procedure basis).

4. OUR PROTOTYPE

The remainder of this paper concerns the Scanning Phase and the prototype implementation we have constructed for the BBN Butterfly Computer. The prototype was written in Modula-2 [14]. Experiments were conducted using the source of the Scanner as input to the scanner.

4.1. The Butterfly Computer

The BBN Butterfly Computer used for these experiments is a first generation moderately sized multiprocessor [3]. The machine we used had about 110 processor nodes. Each node contains its own Motorola 68000 processor and local memory. Nodes are interconnected using a Butterfly switch (logarithmic switching network). Each processor can refer to local memory and to memory on another node (through the Butterfly switch). Remote memory references are transparent to the programmer (after a one-time initial setup). Assuming local memory use, each node is approximately a 0.5 MIPS machine. In the absence of contention a local memory reference is 5 times as fast as a remote memory reference. In our experiments below all of the program, local data areas and stack are contained in local memory. The only non-local memory references are to the distributed global string table.

4.2. Input and Output of Scanner

Our model of the Scanning Phase has a single file (random accessible) of characters as input. In our implementation, a copy of the input file resides in the local memory of each processsing node. We do not consider any issues of disk file input to parallel processors. We also assume that any preprocessor operations (file inclusions, macro expansions, etc.) have been previously

completed. The output from the Scanning Phase consists of:

1) Token Table

The token table is a data structure containing the sequence of tokens (lexemes) found in the input file. Each token is represented by three quanitities: the type of token (e.g, identifier, reserved word **BEGIN**, real literal, operator ':=', and so forth); an indication of where the token was found in the the original file (character index to be used in constructing error or warning messages); and a value interpreted according to the type of the token. The value components for reserved words and operators are empty. The value components for integer and real literals are the bit patterns corresponding to the value of the literal. The value components for a string literal or identifier are references (indices) to the table of strings.

2) String Table

The string table contains the strings corresponding to the user identifiers and character string literals. The string table contains a particular string value only once. Formal equality of two strings may be tested by comparing the indices of the strings in the table. Note that for identifiers, formal equality does not insure symbolic equality (an identifier in one scoping context may refer to a different quantity than the same identifier in a different scoping context). Name/Scope resolution is to performed in a later phase of processing.

3) Bracketing Information

A table containing the indices of tokens which bracket a set of statements. For example the token index of a **LOOP** reserved word and the corresponding **END** reserved word may be stored. For Procedure declarations and Module declarations the token index of the terminating symbol (';' following the procedure name following **END** for procedure declarations and the '.' following the module name following **END** for module declartions). This information is useful in partitioning the sequence of tokens into groups for the parsing phase.

The basic technique involved in the parallel implementation of the Scanner is to logically partition the input file into sequences of characters and give each sequence to a different instantiation of a single process Scanning module with each instantiation running on a separate processor. The results are then merged to form the final output of the scanner.

4.3. Partitioning For Scanning

Our partitioning strategy must be simple to avoid spending more time in partitioning than we save by the parallel scanning of the file. In particular we cannot afford to make a pass throught the entire file before deciding how to partition it.

Let us assume that we have *n* processors available for the Scanner and wish to partition the input file into *n* pieces. If the length of the file were *len* bytes, each piece would be approximately *len/n* bytes. The overhead in merging the results of scanning very small sequences of tokens would probably mean that there is a minimum size piece that we should be willing to scan even if it means using fewer processors. Our estimate (based on our experiments) is that each segment should be a minimum of of a few hundred characters long.

A lexical analysis program is essentially a finite state machine (For Modula-2 , the finite state machine must be augmented with a counter to indicate comment nesting level). We want to start *n* versions of this finite state machine, each handling a different segment of the input file. In all but the first machine, the initial state of the finite state machine is ambiguous. While others [13] have described techniques to handle this ambiguity (basically by communication with the processor on the left), we choose instead to attempt to avoid the ambiguity by making minor changes to the partition points to eliminate (or at least reduce) the state ambiguity at the start of a segment. First of all, we notice that if we choose our partitioning points to be white space (spaces, tabs, end-of-lines etc) we will reduce the possible states to be: *between tokens*; *inside a string literal*; *inside a comment* (of unknown nesting level). The version of Modula-2 that we are dealing with as implemented by Powell[16], allows us to require that string literals not cross line boundaries unless the end-of-line is preceded by a back-slash (\). We may thus force the partitioning points to be at end-of-lines that are not preceded by back-slashes and we reduce our state ambiguity to be either *between tokens* or *inside a comment*. Note that the programming language ADA [5] does not allow comments or string literals to cross line boundaries. Thus a partition of ADA programs at line boundaries would leave an unambiguous state (*between tokens*).

Our partitioning technique for Modula-2 consists of the following. First, we divide the size of the source file *len* by the number of processors available to determine the segment length. We compute the initial starting and ending points of each segment based upon this segment length. We send each set of starting and ending points to a different processor. Each processor (other than the processor handling the initial portion of the input file) positions to its starting point and then skips to the first end-of-line that is not preceded by a backslash. Each processor (including the first) will extend its segment past the nominal ending point until the next end-of-line that is not preceded by a backslash. We still have the ambiguity that we may have partitioned the file either between tokens or inside a comment. For each source segment we keep track of the number of comment start symbols we see that do not have a corresponding comment end symbol, and the number of comment end symbols that do not have a corresponding comment start symbol. When tokenizing a source segment we toss any text between the comment start symbol and either the corresponding comment end symbol or the end of the source segment. If we encounter a comment end symbol without previously seeing a corresponding comment start symbol, we toss away all tokens seen so far in this segment. On completion of the Scanning processes, we can simply look at the counts of comment starts and comment end symbols and eliminate the token tables of entire source segments if appropriate.

For example, assume we have the following program segment. The symbol <*> and description after the symbol <*> are meant to indicate an end-of-line being used as a segment partitioning point.

```
                                        <*> start of segment 1
(* comment1: *)

y := y + 1;                             <*> start of segment 2
x := x - 1;

(* a very
  long                                  <*> start of segment 3
  comment
  extending
  across                                <*> start of segment 4
  many lines
*)


x := x * y;
(* a new
  comment                               <*> start of segment 5
  extending over multiple lines
*)
```

Segment 1 contains an entire comment which would be thrown away by the scanner working on that segment. The tokens returned by that scanner would be just 'y', ':=', 'y', '+' , '1', ';' . The number of comments started (but not terminated) and the number of comments terminated (but not started) in the segment would be zero.

Segment 2 contains some tokens and then an unterminated comment. The tokens returned would be: 'x', ':=', 'x', '-', '1', and ';'. The number of comments started but not terminated would be one and the number of comments terminated but not started would be zero.

Segment 3 would appear to contain the tokens 'comment', 'extending', and 'across'. The number of comments started but not terminated would be zero. The number of comments terminated but not started would be zero.

While processing segment 4 the scanner would find tokens 'many' and 'lines' before encountering the comment terminator. On encountering the comment terminator the scanner would throw away the tokens it had found to that point. The resulting token table for segment 4 would contain just the tokens 'x', ':=', 'x' , '*', 'y', and ';'. The scanner for segment 4 would encounter an unterminated comment. The number of comments terminated but not started would be one. The number of comments started but not terminated would be one.

The scanner for segment five would encounter only a terminated comment. No tokens would be

returned. the count of comments terminated but not started would be one. The count of comments started but not terminated would be zero.

In later processing (before we run the parser) the system would notice (from inspection of the comment counts) that we have a comment starting in segment 2, and terminated in segment 4. Thus all the tokens returned in segment 3 are spurious and would be eliminated. Using the comment counts we would be able to diagnose the errors caused by improper termination of comments.

Unfortunately, this technique is not 100% effective. A pathological case occurs when we have single string quotes within a comment. E. G.

```
(* This is a comment
  that causes problems
  (* because of nesting and single
  quotes (')   endnested *) (')
  end of comment *)
```

If the partitioning point immediately followed the word ***single*** we would have a unterminated comment count of 2 in the earlier segment. We do not know initially that we are inside a comment when we start processing the later segment. Thus the line starting with the word quotes would be scanned as if it were an identifier (***quotes***), a left parenthesis, the string constant ***') endnested *)'*** and a right parenthesis. The comment terminating symbol would not have been noticed and the count of comments terminated in this segment would be one too few. Our current implementation does not handle this condition. The problem occurs only when a comment contains a string literals containing either comment initiators or comment terminators or when the comment contains unbalanced string quotes. This is fairly rare and one technique to handle this situation would be to note if any of the string literals contained in a segment contained the comment initiator or terminator symbols. If such a string literal were to be thrown away because it was contained in a comment then we would rescan the offending segment.

Note we were forced to keep a count of comment starts and ends because Modula-2 allows nested comments. In languages without comment nesting a boolean flag for indicating whether we were processing a comment would be sufficient.

4.4. String Table Construction

Later phases of the compiler should be be able to deal with strings (either from identifiers or character string literals) in a simple manner without need for character by character comparisons. Our implementation involves entering strings uniquely into a table and using the index of the entry as a synonym for the original string. Each string will appear only once within

the table. Thus, later stages of a compiler would determine if two strings are the same by comparing the indices into the string table. However, we also want the individual scanners to work as independently as possible, and the use of a single global string table is potentially a contention point and possible bottleneck that would reduce our parallelism. Our experiments have tested several different approaches to the string table problem.

The first technique is to have each scanner construct its own string table. On completion of all scanners, we merge the string tables and update all the references. This has the property that the scanning stages are truly independent and thus there is no impediment to full parallelism in the scanning stages. Of course the merge and update must be done quickly or we may end up with a bottleneck.

Another approach to have a distributed global string table which is managed by individual string table processes. Thus, a scanning process would, on encountering a new string, compute a hash code from the string and send a message to the string table process corresponding to that hash code. It would continue processing in parallel and the string table process would update the token for the string with the appropriate reference using shared memory techniques. A local table is kept so that if the same string appeared twice within the segment, only one query to a string table process would be generated.

An alternative approach is to eliminate the extra string table processes and to have the scanning processes directly manipulate the distributed global string table (using shared memory and appropriate synchronization locks). We have a single global string table and a global hash table for the string table which is distributed into the different processors memories. Thus we have a two level hash algorithm. The first hash of a string determines which processor's memory would contain the string. The second hash is used to determine a hash bucket on that processor. A linked list for each bucket is used to contain the strings that hashed to that bucket. A mutual exclusion lock is used for each bucket to avoid problems with simultaneous writers and readers refering to the bucket. A partial copy of the global string table (only those strings hashing to the particular processor) is kept on the appropriate processors. A local copy of strings seen by the individual scanner is used to avoid extra references to the global table.

The final approach considered is the same as that described in the previous paragraph except that no local copies are kept by the individual scanners.

During the course of our prototype construction we have implemented all of the methods described above. The fastest of all these approaches on our Butterfly Computer is the last one, use of a distributed global hash table with no local copies. At first, this seemed counter-intuitive as the global table would appear to be a potential bottleneck. It turns out not to be a bottleneck because the table itself is distributed and the probability of simultaneous access to the parts of the table on a single processor from multiple processors is relatively small. The Butterfly's memory structure is set up so that there is enough bandwidth to handle small numbers of

references from remote processors to a node's local memory without noticeable performance degradation on the local node. In practice the memory contention bottleneck did not appear until a few dozen processors were simultaneously in use (our belief is that this is not so much memory contention as switch contention). The other approaches were slower because of the overhead of message traffic, or the overhead in maintaining separate local copies of part of the String Table.

4.5. Bracketing Tables

The compilation system would like to partition the token stream corresponding to complete syntactic entities (such as procedure declaration, statement blocks, if-then-else statement etc) before instantiating the parallel parsers. It is convenient to have the scanning phase determine where such entities start and terminate within the token stream. The syntactic entities are very language dependent. In Modula-2 we have determined that the entities of interest are:

IMPLEMENTATION modules (opened by **IMPLEMENTATION MODULE**, closed by sequence **END** *ident* ".")
PROGRAM modules (opened by **MODULE**, closed by sequence **END** ident ".");
DEFINITION modules (opened by **DEFINITION**, closed by sequence **END** *ident* ".");
MODULE declarations (opened by **MODULE** closed by **END** *ident* ";");
PROCEDURE declarations (opened by **PROCEDURE** closed by **END** *ident* ";")
RECORD-END types (opened by **RECORD** closed by **END**)
FOR-END statements (opened by **FOR** closed by **END**)
CASE-END statements (opened by **CASE** closed by **END**)
IF-END statements; (opened by **IF** closed by **END**)
LOOP-END statements; (opened by **LOOP** closed by **END**)
REPEAT-END statements; (opened by **REPEAT** closed by **END**)
WHILE-END statements; (opened by **WHILE** closed by **END**)
WITH-END statements;(opened by **WITH** closed by **END**)

We note that each of these constructs is terminated by a sequence (length <= 3) of tokens which starts with the reserved word END. Each entity starts (with the single exception of IMPLEMENTATION modules) with a single well known reserved word. Our technique is to have each of the scanning modules construct a table of all the bracketing information encountered. The table is stored as a sequence of bracket names (such as PROCEDURE or END) and the token index corresponding to the bracketing token.

Our current implementation assumes that the parsing stages will only want to deal with complete units of IMPLEMENTATION modules, PROGRAM modules; DEFINITION modules; MODULE declarations; and PROCEDURE declarations. Thus, we have the individual scanning processes remove those pairs from their bracket tables corresponding to the statements or RECORD types.

5. EXPERIMENTS

We executed the scanner on various sized files and measured the performance. The times given below do not include the initial set up time. The set up time would include copying input file into local memories, setting up memory mapping, starting up the individual processes on the nodes; inserting the Predeclared identifiers (such as INTEGER, REAL and so forth) into the string table. The times do include the messages needed to tell each processor what portion of the input it was to scan. The times do not include the finalization times (copying all the tokens to a single area of memory, bracket table analyses and so forth).

We wish to measure the inefficiencies caused by three factors: contention; initialization overhead; and the requirement that processors finishing early must idle until all processors complete.

The effective number of processors is computed as:

$$\text{Effective}(n) = \text{Time}(1)/\text{Time}(n)$$

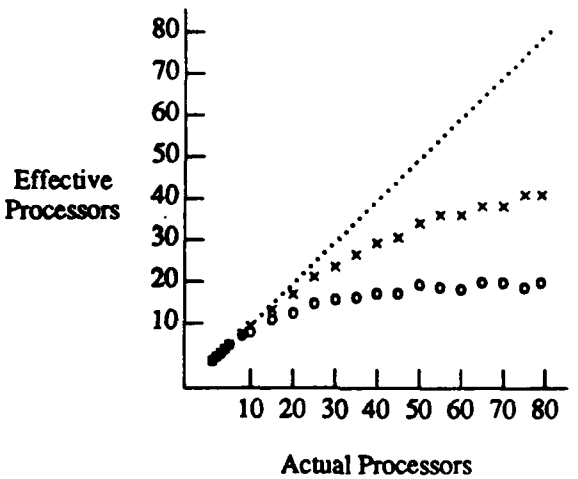
The average effective number of processors is derived by not including the idle time a processor. This is accomplished by adding up the times to complemetion of the individual processors and dividing by the number of processors to get an average time.

For the purposes of exposition we provide the results of running five different files through the system with various number of processors. Because of a defect in our Modula compiler which prevented large token tables per processornode, we were unable to scan large files on a small number of processors (because there would be too many tokens in the token table).

5.1. Example 1 The first example is of a relatively small file. The non-uniform partitioning is very significant (compare the average effective processors with the effective number of processors). The overhead of startup (about 4-5 milliseconds) becomes very significant above about 30 processors where the segment size gets small (less than 100 bytes).

File: ScanNumbers.mod
Size: 3170 bytes
Tokens: 615

Processors	Time (millisec)	Effective Processors	Avg. Time (millisec)	Avg. Effective Processors
1	614	1.0	613	1.0
2	322	1.9	314	2.0
3	227	2.7	213	2.9
4	165	3.7	158	3.9
5	131	4.7	126	4.9
8	87	7.1	81	7.6
10	78	7.9	66	9.3
15	57	10.8	46	13.3
20	50	12.3	36	17.0
25	42	14.6	29	21.1
30	39	15.7	26	23.5
35	38	16.1	23	26.6
40	36	17.1	21	29.2
45	36	17.1	20	30.7
50	32	19.2	18	34.1
55	33	18.6	17	36.1
60	34	18.1	17	36.1
65	31	19.8	16	38.3
70	31	19.8	16	38.3
75	33	18.6	15	40.9
79	31	19.8	15	40.9

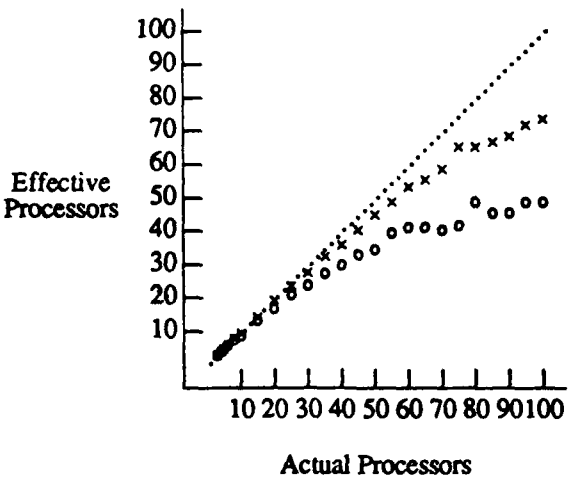


The line of dots represents an ideal linear speedup; the line of crosses indicates the Average Effective Processors; and the line of circles indicates the Effective Processors.

5.2. Example 2 This next example also illustrates the problems of nonuniform partitioning (look at the relative effective number of processors above 50 real processors). The problems of contention and/or small segment size shows up above about 60 real processors (look at the avg effective processors).

File: Scanner.mod
Size: 16568 bytes
Tokens: 2202
Extrapolated time for a single processor is 2884 milliseconds.

Processors	Time (millisecs)	Effective Processors	Avg Time (millisecs)	Avg. Effective Processors
3	978	2.9	964	3.0
4	747	3.9	727	4.0
5	615	4.7	584	4.9
6	500	5.8	486	5.9
8	383	7.5	365	7.9
10	323	8.9	297	9.7
15	217	13.3	201	14.3
20	172	16.8	149	19.4
25	137	21.1	122	23.6
30	121	23.8	104	27.7
35	105	27.4	88	32.7
40	96	30.0	80	36.1
45	87	33.1	71	40.6
50	83	34.7	64	45.1
55	73	39.5	59	48.9
60	70	41.2	54	53.4
65	70	41.2	52	55.5
70	71	40.6	49	58.9
75	69	41.8	44	65.5
80	59	48.9	44	65.5
85	63	45.8	43	67.1
90	63	45.8	42	68.7
95	59	48.9	40	72.1
100	59	48.9	39	73.9

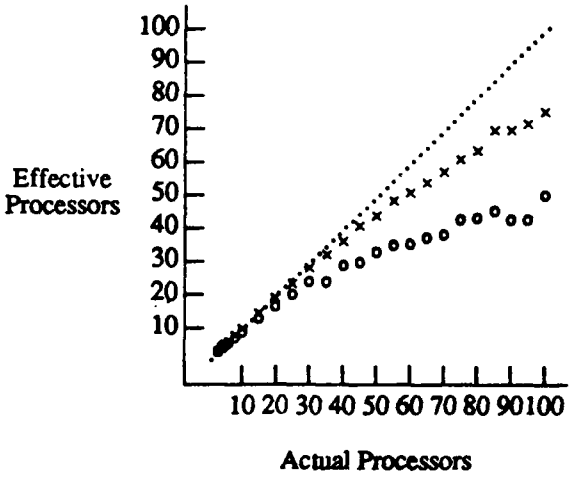


Again the lines of dots represent a linear speedup, the circles represent effective number of processors and the crosses represent effective processors when idle time is eliminated.

5.3. Example 3 The next example is again of a moderate sized file. Its results are comparable to the previous example.

File:MultiScanner.mod
Size: 16558 bytes
Tokens: 2202
Time for a single processor extrapolated to be 2863 milliseconds

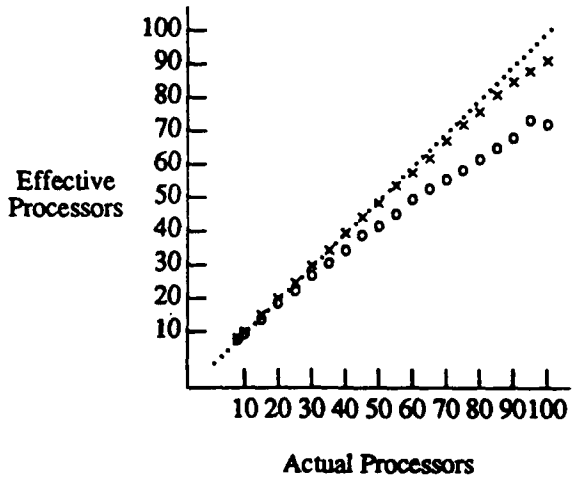
Processors	Time (millisec)	Effective Processors	Avg. Time (millisec)	Avg. Effective Processors
3	1012	2.8	957	3.0
4	739	3.9	725	3.9
5	628	4.6	578	5.0
6	528	5.4	484	5.9
8	406	7.1	365	7.8
10	328	8.7	290	9.9
15	221	13.0	197	14.5
20	173	16.5	150	19.1
25	141	20.3	123	23.3
30	120	23.9	102	28.1
35	120	23.9	89	32.2
40	99	28.9	79	36.2
45	96	29.8	70	40.9
50	87	32.9	65	44.0
55	82	34.9	59	48.5
60	81	35.3	56	51.1
65	77	37.2	53	54.0
70	75	38.2	50	57.3
75	67	42.7	47	60.9
80	66	43.4	45	63.6
85	63	45.4	41	69.8
90	67	42.7	41	69.8
95	69	42.7	40	71.6
100	57	50.2	38	75.3



5.4. Example 4 The following example is of the maximal size file handled by the current implementation (again due to defects in the Modula compiler). The file is about 64,000 bytes long. This file is somewhat artificial in that the file is really a concatenation of several copies of a much smaller file. Thus the string table contention is less than would be expected because the ratio of string lookups to string insertions is higher. Again the biggest inefficiency is due to the non-uniform partitioning. Small segment overhead, and string table contention are not as significant as we can see by the Avg Effective Processors column.

File: Hugefile.mod
Size: 63989 bytes
Tokens: 9130
Time for a single processor extrapolated to be 11010 milliseconds

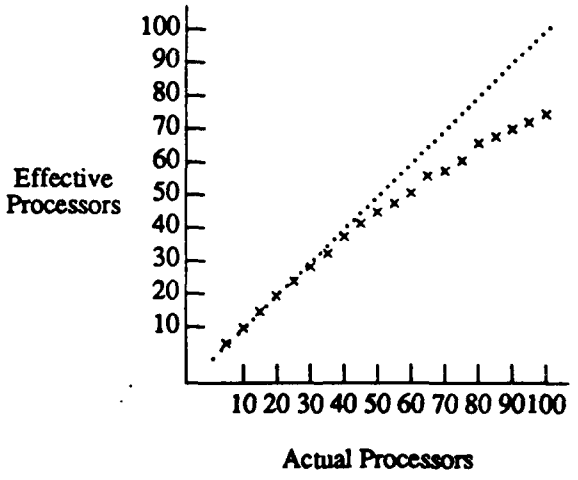
Processors	Time (millisec)	Effective Processors	Avg Time (millisec)	Avg. Effective Processors
8	1455	7.6	1380	8.0
10	1171	9.4	1097	10.0
15	804	13.7	735	15.0
20	597	18.4	551	20.0
25	494	22.3	444	24.8
30	410	26.9	370	29.8
35	360	30.6	319	34.5
40	319	34.5	279	39.5
45	282	39.0	249	44.2
50	264	41.7	227	48.5
55	243	45.3	205	53.7
60	222	49.6	191	57.6
65	208	52.9	178	61.9
70	198	55.6	164	67.1
75	189	58.3	153	72.0
80	179	61.5	145	75.9
85	169	65.1	136	81.0
90	162	68.0	130	84.7
95	150	73.4	125	88.0
100	153	72.0	121	91.0



5.5. Example 5 To get an estimate on the problems with string table contention we reran a moderately sized file with string table lookups and insertions disables. Thus there should have been no contention at all. The problems of non-uniform partitioning still appear as well as the overhead for small segments.

File: MultiScanner.mod
Size: 16897 bytes
Tokens: 2256
Time for a single processor extrapolated to be 2230 milliseconds

Processors	Time (millisec)	Effective Processors	Avg Time (millisec)	Avg. Effective Processors
5	469	4.8	449	5.0
10	248	9.0	228	9.8
15	166	13.4	153	14.6
20	128	17.4	115	19.4
25	108	20.6	94	23.7
30	92	24.2	79	28.2
35	90	24.8	69	32.3
40	75	29.7	60	37.2
45	68	32.8	54	41.3
50	67	33.3	50	44.6
55	63	35.4	47	47.4
60	59	37.8	44	50.7
65	54	41.3	40	55.8
70	55	40.5	39	57.2
75	51	43.7	37	60.3
80	50	44.6	34	65.6
85	49	45.5	33	67.6
90	47	47.4	32	69.7
95	52	42.9	31	71.9
100	44	50.7	30	74.3



Note the best effective processors value for 100 actual processors we could hope for would be 84.9 (assuming 4 milliseconds overhead) If there were 5 milliseconds overhead the best would be 81.8 effective processors.

6. Conclusions

Overall the experiments were a success in that we demonstrated that dozens of processors (on the order of 40 for normal sized source files) could be efficiently used to perform the lexical analysis stage of a compiler.

Our analysis of the experiments shows several problems which tend to limit the amount of parallelism that may be obtained for many classes of problems. The problems are contention for shared resources; non-uniform partitioning; and partitioning overhead.

In the our example, the largest loss of parallelism is most commonly due to the non-uniform partitioning of the input file. The loss of parallelism is caused by our requirement that each phase of compilation be completed before the next is started. Since the variability of processing time for different data (even though approximately the same length) is so large (seen as much as a factor of 3) we tend to have processors totally idle for significant amounts of time. If we could have these idle processors work on other phases of the compilation (or on other problems) this inefficiency could be eliminated.

Contention shows up when we have large numbers of processors doing updates on the global symbol table. Our conjecture is that this is more from switch contention than memory or string table lock contention.

Partitioning overhead consists of the extra work needed to be done because we split up the program. This consists of at least several components including: the actual partitioning code executed; the code executed to merge the outputs; and the common initializations and finalizations that must now be done on each processor where normally they would only be done once. Partitioning overhead becomes significant when the granularity of processing is quite small. In the scanning example, we find significant partitioning overhead when the size of individual segments is less than a few hundred characters. This indicates that for modules of a few thousand bytes in length, at most a few dozen processors may be efficiently exploited.

References

[1] A Aho, R. Sethi, and J Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] BBN Advanced Computers Inc. *Chrysalis Programmers Manual, Version 3.0.* May 1987.

[3] BBN Laboratories, *Butterfly Parallel Processor Overview*, BBN Report # 6148, Version 1. Cambridge Massachusettes, March 1986.

[4] A. de Boor. *PMake -- A Tutorial*, Computer Science Department, The University of California, Berkeley, July 1988.

[5] Department of Defense, *Reference manual for the ADA programming Language.* July 1980.

[6] P.H. Enslow, *Multiprocessor Organization- A Survey, ACM Computing Surveys*, Vol 9, No. 1 Mar 1977 p 103-129.

[7] A. Gottlieb et al. *The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer.* IEEE Transactions on Computers, Volume C-32 No. 2, Feb 1983 p 175 -189.

[8] A. K. Jones, P. Schwartz, *Experience Using Multiprocessor Systems - A Status Report*, ACM Computing Surveys Volume 12. No 2 June 1980 p 121-166.

[9] N. Lincoln, *Parallel Programming Techniques for Compilers.* SIGPLAN Volume 5, No 10 1970.

[10] D. Lipkie, *A Compiler Design for Multiple Independent Processor Computers.* PhD thesis. University of Washington, Seattle, 1979.

[11] J. A. Miller and R. J. Leblanc. *Distributed compilation: A case study* in IEEE Proceedings of the 3rd International Conference on Distributed Computing. October 1982.

[12] R. Loka, *A Note on Parallel Parsing*, SIGPLAN Notices Jan 1984 p 57-59.

[13] M. D. Mickunas, R. M. Schell, *Parallel Compilation in a Multiprocessor Environment.* Proc. ACM Annual Conference, p 241-246, 1978.

[14] T. Olson, *Modula-2 on the BBN Butterfly Parallel Processor.* Butterfly Project Report 4, Department of Computer Science, The University of Rochester Jan 1986.

[15] G. F. Pfister et al, *The IBM research parallel processor prototype (RP3): Introduction and architecture* in Proc. of the IEEE 1985 International Conference on Parallel Processing. IEEE Press, New York 1985 p 764-771.

[16] M. L. Powell, *Using Modula-2 with Unix C and Berkeley Pascal*, DEC Western Reaseach Laboratory, May 1984.

[17] C. L. Seitz, *The Cosmic Cube.* Comm. ACM Vol 28, No 1 (Jan 1985) p 22-23.

[18] R. Teitelbaum, and T. Reps, *The Cornell program synthesizer: a syntax directed programming environment*, Comm. ACM Vol 24 No. 9 (Sept 1981) p 563-573.

[19] W. Teitelman, L. Masinter, *The Interlisp Programming Environment*, ComputerVol 14 No. 4 p 25 -33.

[20] M. Vandervoorde, *Parallel Compilation on a Tightly Coupled Multiprocessor*. Research Report 26, Digital Equipment Corporation System Research Center, Palo Alto, California, March 1988.

[21] N. Wirth, *Programming in Modula-2*, 2nd Ed. Springer-Verlag 1982.

[22] W. Wulf et al, *The Design of an Optimizing Compiler*, American-Elsevier 1975.